Acceleration of Acoustic Emission Signal Processing Algorithms using CUDA Standard

Lubomir Riha[a,b], Radislav Smid[a]

[a] *Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Measurement, Prague, Czech Republic*
[b] *Bowie State University, Bowie, MD, USA*

Abstract

Offline processing of acoustic emission (AE) signal waveforms recorded during a long-term AE monitoring session is a challenging problem in AE testing area. This is due to the fact that today's AE systems can work with up to hundreds of channels and are able to process tens of thousands of AE events per second. The amount of data recorded during the session is very high.

This paper proposes a way to accelerate signal processing methods for acoustic emission and to accelerate similarity calculation using the Graphic Processing Unit (GPU). GPU-based accelerators are an affordable High Performance Computing (HPC) solution which can be used in any industrial workstation or laptop. They are therefore suitable for onsite AE monitoring.

Our implementation, which is based on Compute Unified Device Architecture (CUDA), proves that GPU is able to achieve 30 times faster processing speed than CPU for AE signal preprocessing. The similarity calculation is accelerated by up to 80 times. These results prove that GPU processing is a powerful and low-cost accelerator for AE signal processing algorithms.

Keywords:
CUDA standard, Multicore Platforms, GPU, Acoustic Emission Testing

1. Introdution

During long-term monitoring on pressure vessels, today's Acoustic Emission (AE) systems with more than one hundred channels are able to process tens of thousands of AE events per second and record them to data storage. This means that large data sets can be recorded in a session, and need to be processed. This is a challenging task for a single, dual or quad-core processor. It is therefore advantageous to use a High Performance Computing (HPC) approach to significantly reduce the processing time from hours to minutes.

Email addresses: rihal1@fel.cvut.cz (Lubomir Riha), smid.fel.cvut.cz (Radislav Smid)

This paper introduces several multicore architectures suitable for parallel implementation of signal processing algorithms. Main attention is paid to Graphic Processing Unit (GPU) based accelerators and its Compute Unified Device Architecture (CUDA) programming model, because this standard was chosen to improve the performance of the algorithm, as described later in this paper.

GPU-based accelerators were chosen because they can be used in any industrial workstation or laptop that is suitable for onsite AE monitoring. They are also an affordable High Performance Computing (HPC) solution that deliver supercomputing performance to a single workstation.

1.1. Multicore Platforms for Digital Signal Processing

General purpose multicore platforms are no longer dominant only in the supercomputing area. They are now accepted in all segments of science and industry, including signal processing. The acceptance of multicore platforms has been forced by the need for more performance while maintaining the simplicity and effectiveness of programming a general purpose CPU.

The history of parallel processors goes back to the Solomon computer in 1960. Wider use was restricted because they were difficult to programm. In the past years, many HPC companies built parallel machines. Most of them were unsuccessful because their programming was complicated and the number of customers was also limited due to their high price.

The major chip manufactures have recently started offering parallel machines or single chips with multiple cores, called multicore processors, to continue the raw performance growth that was expected from Moore's law scaling without being overwhelmed by power consumption. This is no longer possible with single core processors, because the power consumption of such solutions grows more rapidly than the clock speed. A multicore processor can be scaled by adding more cores rather than by increasing the frequency, and this means slower growth of power consumption.

The prediction from the ITRS roadmap [1] states that by 2022 performance should grow 300x times. To be able to deliver such performance, future chips will need to have 100x more cores than current multicore processors.

1.2. Commercial multicore platforms

In the past few years many different multicore architectures have been produced by major companies for the commercial market.

Table 1 shows a list of some general purpose processors (Phenom, Core i7, Niagara and Atom). For example, the Intel Core i7 is a high performance general purpose processor. It is built from up to 8 cores, where each core contains a 128bit SIMD unit useful for data-parallel operations such as vector operations. Like most Intel processors, it supports the CISCx86 ISA architecture. Beyond the ISA definition, other extensions are added to improve multimedia performance, such as MMX, MMX2 and SSE1-4 [3]. The

Name	ISA	Number of cores	Cache	Power [W]	Frequency [GHz]	Operations per clock
AMD Phenom	x86	4	IL1, DL1 64kB/core L2 256kB/core L3 2-6 MB	140	2.5-3.0	12-48
Intel Core i7	x86	2-8	IL1, DL1 32kB/core L2 256kB/core L3 8 MB	130	2.66-3.33	8-128
Sun Niagara	SPARC	8	IL1 16kB, DL1 8kB/core; L2 4MB	60-123	0.9-1.4	16
Intel Atom	x86	1-2	IL1, DL1 32kB/core L2 512kB/core	2-8	0.8-1.6	2-16
ARM Cortex A9	ARM	1-4	IL1, DL1 (16,32,64)/core L2 up to 2MB	1W	N/A	3-12

Table 1: General purpose desktop CPUs and processors for embedded platforms [2]

memory system is typical for general purpose processors with few cores. The cache system is fully coherent with large standard caches.

Name	ISA	Number of cores	Cache	Power [W]	Frequency [GHz]	Operations per clock
AMD Radeon R700	N/A	160 see (1)	16kB local cache/SIMD block (16 cores)	150	0.75	800
NVidia G200	PTX 1.0	240 see (2)	16kB local cache/SIMD unit (8 cores)	183	1.2	240
Nvidia Fermi	PTX 2.0	448 see (3)	48kB L1/SIMD unit (32 cores) 768kB L2	225	1.15	448
IBM CELL	POWER	1 PPU 8 SPU	PPU: 32kB IL1 and 32kB DL1; 512kB L2 SPU: 256kB local store	100	3.2	72

Table 2: High performance platforms which are available as accelerators for PC workstations, note (1): 16 cores per SIMD block, 10 blocks; note (2): 8 cores per SIMD unit, 30 SIMD units; note (3): 32 cores per SIMD unit, 14 SIMD units, [2]

Table 2 shows the set of architectures specialized for high performance computing. Their main goal is high performance, so that high power consumption is not a major issue in this area. The first two platforms (Radeon R700 and G200) are GPUs (Graphic Processing Unit) [4, 5] and were originally designed for computer graphics. However, both of them are able to execute general purpose programs written in C language with hardware-specific extensions. The high performance of the architectures is provided by

hundreds of cores.

A Cell processor is different from conventional processors inside. The Cell is a heterogeneous chip multiprocessor that consists of an IBM 64-bit Power Processing Element (PPE), augmented by eight specialized co-processors based on a novel single-instruction multiple-data (SIMD) architecture called Synergistic Processor Element (SPE), which can be used for data intensive processing [6]. The Power processor governs the allocation of resources, including the SPEs and other resources to the various OS partitions. The Cell is mainly a distributed memory processor, unlike Power processors. SPEs are equipped with scratchpad memory, referred to as the Local Store (LS), and a Memory Flow Controller (MFC) to perform Direct Memory Access (DMA) transfers of code and data between the system memory and the Local Store [7]. SPEs are bound together by a fast internal bus, the Elemental Interface Bus (EIB). A built-in dual channel memory controller is included. Communication with the rest of the system is provided by the FlexIO bus. This interface also allows high speed, chip-to-chip communication between different Cell processors.

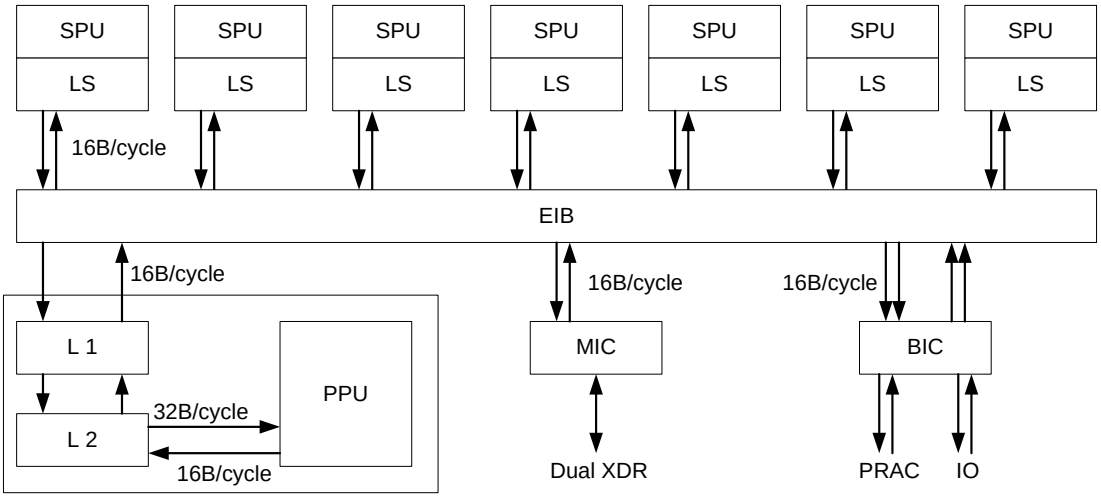

Figure 1: Block diagram of the CELL BE architecture

All three high performance platforms (from Table 2) can be used as accelerators in the form of a PCI-Express card in a PC based workstation, which is the ideal platform for AE testing. GPU-based accelerators can also be used in laptops. This provides the additional mobility necessary for onsite monitoring.

2. NVidia GT200 and Fermi Architecture

The G80 was the first GPU able to be programmed with C programming language with the CUDA extension. This generation was introduced in November 2006 and brought the single-instruction multiple-thread (SIMT) execution model, where multiple independent threads execute concurrently using a single instruction. In June 2008, the second generation unified architecture – GT200 – was introduced [4]. This

generation increased the maximum number of CUDA cores from 128(G80) to 240 and added the double precision floating support that is important for scientific and high performance needs. The architecture of GT200 is specifically aimed at data dominated applications but introduces performance penalties when branches and random memory access are performed by the program.

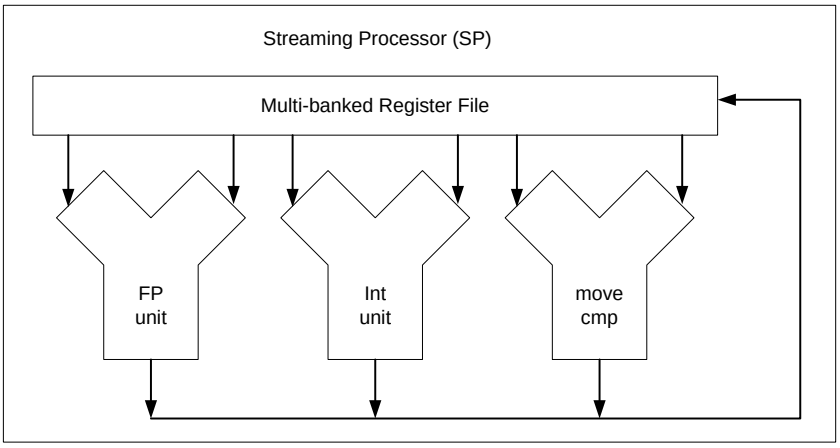

Figure 2: Block diagram of the streaming processor in GT200 architecture

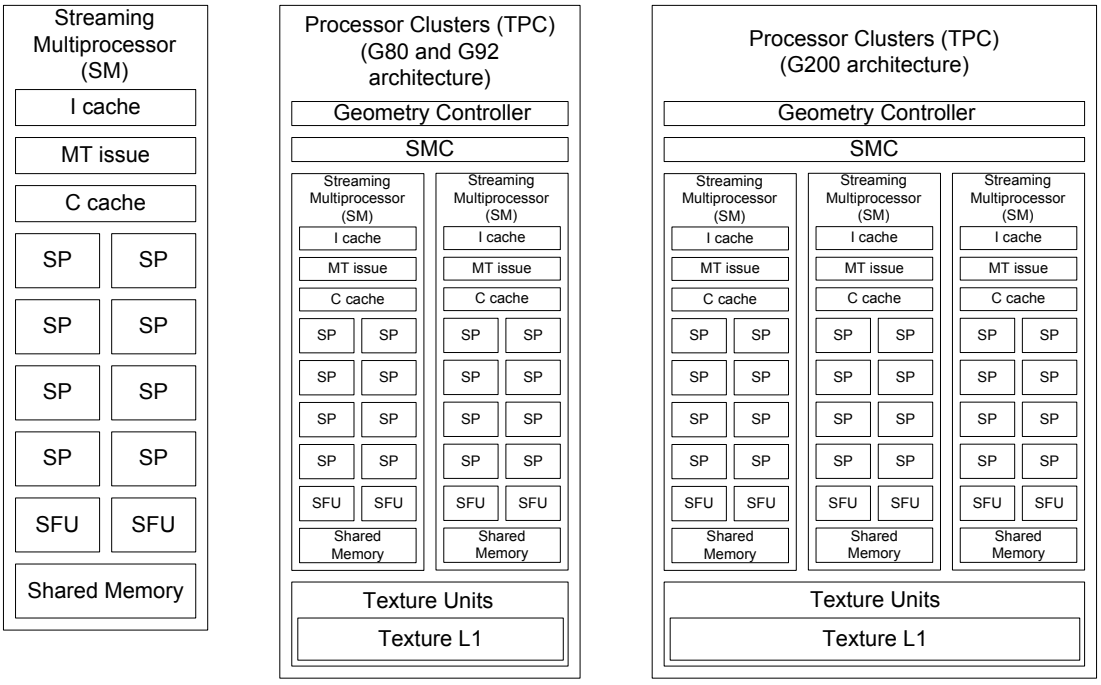

Figure 3: Block diagram of the streaming multiprocessor (left) and the processor clusters (TPC) of G80 (middle) and GT200 (right) architectures

The GT200 architecture contains 240 cores (called streaming processors (SP) or CUDA cores), which are divided into clusters of 24 cores (called texture clusters or processor clusters or TPCs). In each cluster, a group of 8 cores shares 16kB of local store memory. These 8 core units are called the streaming multiprocessor (SP). All 24 cores in one cluster are controlled in SIMD (single-instruction multiple-data) manner, which means that each core executes the same instruction from different threads. In comparison with the SIMD processor, the GT200 architecture has two significant advantages: each core is capable of branching (if this happens, all the cores must execute both paths of the branch and only keep the part they would have followed) and the cores are able to access memory independent addresses in the memory. However, this kind memory access significantly slows down the data transfer rate between GPU main memory and processors. These two features put GT200 somewhere between SIMD and MIMD (multiple-instruction multiple-data).

The memory system is designed for the data-parallel programming approach. The local stores are small (GT200 contains caches, but these are used for raster graphic computing, and not for general purpose computing) in comparison with the cache sizes of general purpose processors (Intel Core i7 has more than 8 MB of cache). The small onchip memory space is balanced by high speed low latency DRAM.

The performance of GT200 was compared with that of Intel Core i7 2.66GHz four cores, CELL BE and FPGA Virtex 4 and Virtex 6 for calculating the discrete Fourier transform [8]. The FPGA Virtex 4 delivers performance up to 40 Gflops/s for DFT (size 256). The Virtex 6 performance is twice higher. GT200 is able to deliver throughput up to 250 Gflops/s for DFT size 512. For bigger DFT sizes the performance decreases to 100 Gflops/s. The peak performance for CELL BE is around 40 Gflops/s (DFT size 16k), which is similar to the performance of Core i7 35Gflops/s (DFT size 4k). All results are for single precision. A comparison for different DFT sizes is given in [8].

The Fermi is the next generation of NVIDIA's GPU. The architecture is designed to contain up to 512 CUDA cores (the chip contains 3 billion transistors). The cores are organized into 16 streaming processors of 32 cores each. GPU has six 64-bit memory partitions supporting up to 6 GB GDDR5 DRAM memory (the memory interface is 384 bits wide).

This brings some significant improvements necessary for general-purpose programming:

- double precession performance has been improved – up to 8 times faster than GT200
- ECC support – allows safe data store in GPU memory
- true cache hierarchy – contains L1 (configurable 16 or 48kB) and L2 cache 768kB
- more shared memory – configurable 16kB or 48 kB)

A block diagram of the streaming multiprocessor (MS) is shown in Figure 5. It shows that one SM contains:

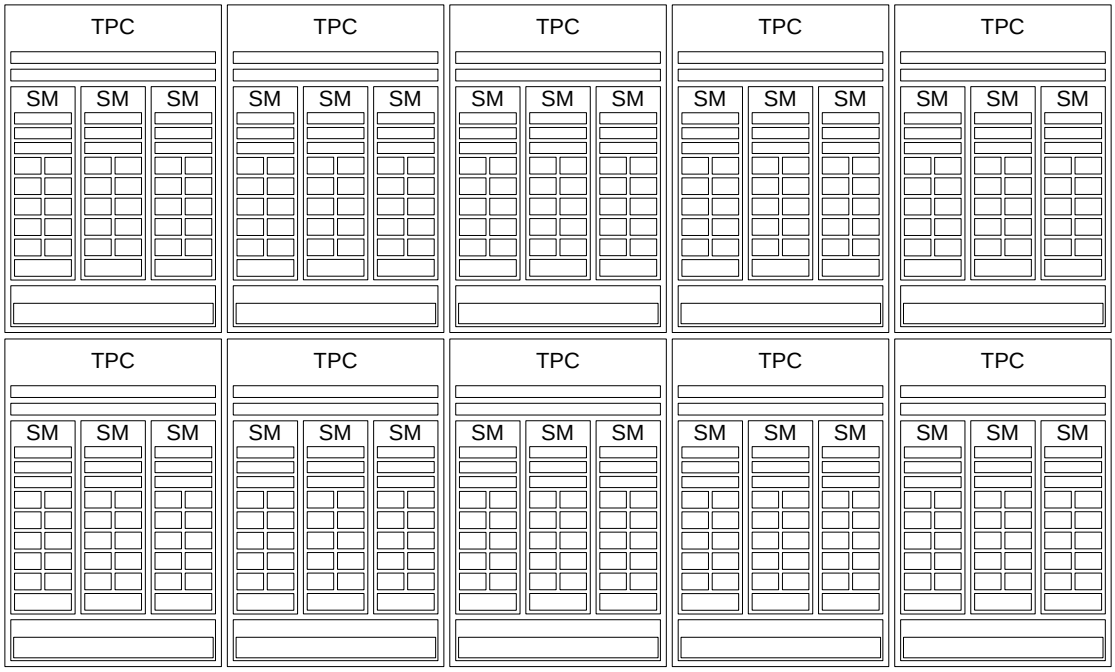

Figure 4: The architecture of GT200 with all 10 TPCs

- 32 CUDA cores, each of which has:
  - one integer arithmetic logic unit (ALU) - supports full 32-bit precision for all instructions and 64-bit precision for extended precision operations
  - one floating point unit (FPU) – provides fused multiply-add (FMA) for both double and single precision
- 16 load and store units – calculates the source and destination addresses in cache or DRAM for 16 threads per clock
- 4 special function units (SFU) – executes transcendental instructions such as sin, cosine, square root, etc. Each SFU executes one instruction per thread, per clock. The 32 threads are executed in 8 clocks.
- a dual warp scheduler - SM schedules threads in groups of 32 parallel threads called warps. The dual warp scheduler allows two warps to be issued and executed concurrently by selecting two warps, and issues one instruction from each warp to a group of sixteen cores, sixteen load/store units, or four SFUs.
- 64 KB configurable shared memory and L1 cache – The shared memory is fast on chip memory into which all threads in one block can access using a special read and write instruction. The GT200 has only 16kB of shared memory. However, in Fermi architecture the total amount of memory per SM

is 64kB, which can configured as 48kB of shared memory and 16kB of L1 cache or 16kB of shared memory and 48kB of L1 cache.

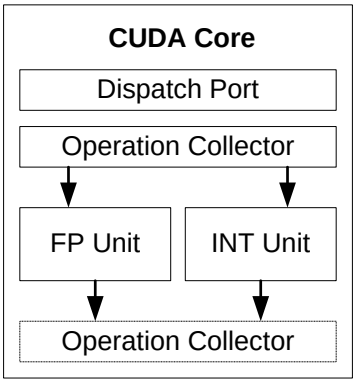

Figure 5: Single CUDA core of Nvidia Fermi architecture (from [9], courtesy of NVIDIA Corp.)

The Fermi architecture supports the new Parallel Thread eXecution (PTX) instruction set. PTX is a low level virtual machine and ISA designed to support the operations of parallel thread architecture. At program install time, PTX instructions are translated to machine instructions by the GPU driver [9].

3. General Purpose Programming on GPUs

Three main technologies are used for general purpose programming on GPUs:

- Compute Unified Device Architecture (CUDA), developed by NVIDIA [4, 10],
- BROOK+ from Advanced Micro Devices (AMD) [11],
- Open Computing Language (OpenCL), maintained by Khronos Group [12].

In this section, OpenCL and Brook+ are briefly described, while the CUDA programming model is described in the next section.

Brook+ is based on BrookGPU, developed at Stanford University [13]. It is designed for AMD GPUs as a data-parallel (SIMD) extension to standard ANSI C with a special compiler. In the Brook+ programming model, GPU consists of a set of thread processors. Each thread processor can execute up to 5 scalar operations per cycle. The two key elements in Brook+ language are stream and kernel. Stream is a collection of data elements of the same type that can be operated on in parallel, while kernel is a parallel function that operates in streams and executes them on multiple thread processors.

OpenCL is an open standard for programming heterogeneous collections of CPUs, GPUs or other devices. This means that the same program can run GPU from NVIDIA or ATI. It is a framework for parallel

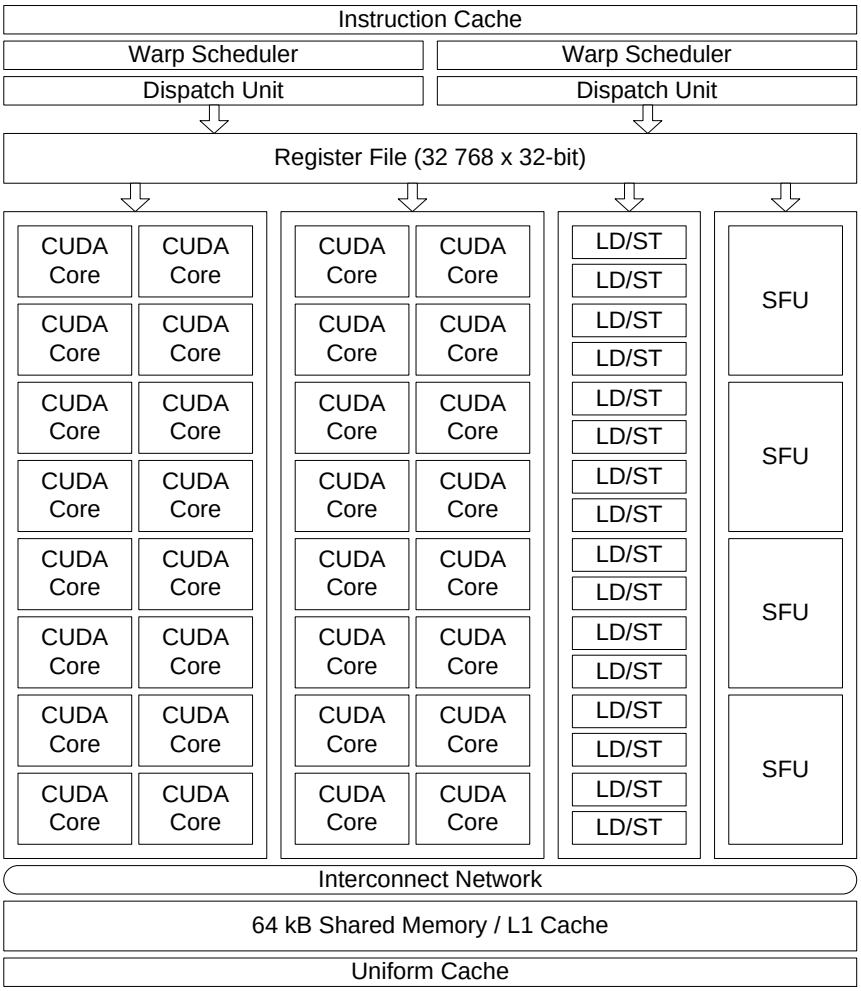

Figure 6: Streaming multiprocessor of Nvidia Fermi Architecture (from [9], courtesy of NVIDIA Corp.)

computing which includes language, API, libraries and the runtime system. The platform model consists of the host (in most cases a general purpose CPU) and one or more compute devices (here GPU(s)). The compute device is further divided into one or more compute units (CU), and these contain one or more processing elements (PE). The data processing is done by processing elements. The program is divided into the host program and one or more kernels which are executed on the device.

Over the past few years, many parallel solutions have been developed for Matlab, such as: the Parallel Computing Toolbox and the Matlab Distributed Computing Server from Mathworks [14]; Star-P from Interactive Supercomputing Inc. [15]; pMatlab from MIT Lincoln Laboratories [16] with bcMPI [17]. These solutions are mostly for general purpose multicores or clusters of workstations. GPU acceleration is also supported using the Matlab plug-in for CUDA [18] or various toolboxes (GPGPUmat [19], gpulib [20] or Jacket [21]). A thorough review of multicore acceleration for Matlab is presented in [22].

4. GPU Programming using CUDA C Extension

In the CUDA programming model, the GPU is a compute **device** and serves as a coprocessor with its own DRAM memory to the main CPU, which in CUDA terminology is called **host**. The host code manages GPU memory. Therefore, before a new kernel is executed the host must allocate space in GPU memory and copy the input data from the host memory to the device memory. The host also sends commands to launch the kernels, which means that the host also sets up the kernel configuration in terms of grid and block size. When kernel execution is finished, the host transfers the data from device memory back to the host main memory and performs further processing.

The data parallel portions of an application are executed on the device as kernels which run in parallel in many threads. In other words the kernel is executed by an array of threads. All threads run the same code, but each thread has its own unique ID (`threadID`) which is used to compute memory addresses and make control decisions using branches.

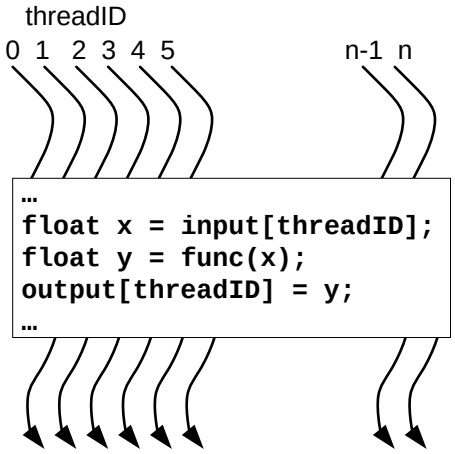

Figure 7: SIMD operation performed by multiple threads in one block

Kernels can be seen as a function which is executed on a device. GPU threads are extremely lightweight, and a GPU needs thousands of threads to achieve full efficiency.

The kernel is executed as a grid of thread blocks (see the right block diagram in Figure 8). A thread block is a batch of threads that can cooperate with each other by sharing data through shared memory and synchronizing their execution. The threads from different blocks cannot be synchronized and do not have access to the same shared memory. In the kernel the thread, block and grid information are accessible via the following constants:

- `threadID` – index of the thread in a block (one, two or three dimensional addressing `threadID.x`, `threadID.y` and `threadID.z`)

- `blockID` – index of the block in the grid (one or two dimensional addressing `blockID.x` and `blockID.y`)
- `blockDim` – size of the block or, in other words, the number of threads in the block (one or two dimensional). The maximum number of threads in a block is 512.
- `gridDim` – dimension of the grid in the blocks (one or two dimensional `gridDim.x` and `gridDim.y`). The maximum number of blocks in the grid is 65535.

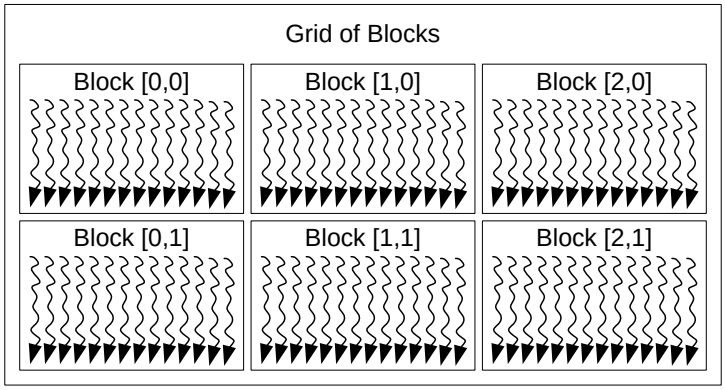

Figure 8: Block diagram of a grid of blocks in the CUDA programming model (from [4], courtesy of NVIDIA Corp.)

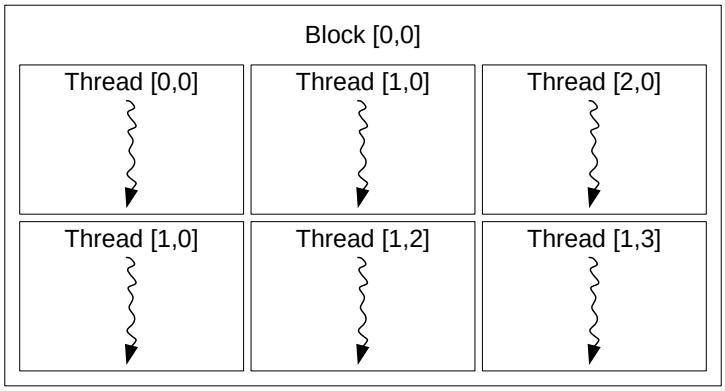

Figure 9: Block diagram of a block of threads in the CUDA programming model (from [4], courtesy of NVIDIA Corp.)

The memory model of a CUDA device contains multiple memory spaces:

- registers
  - accessible by one thread only
  - physically residing on a chip

  - the data lifetime is the same as the thread lifetime

- local memory
  - accessible by one thread only
  - physically residing in device DRAM memory
  - the data lifetime is the same as the thread lifetime

- shared memory
  - accessible by all threads in a block
  - physically residing on chip
  - the data lifetime is equal to the block lifetime

- global device memory – primary resource used for communication between host and device
  - accessible by all threads (GPU) and the host (CPU)
  - the data lifetime is from allocation by the host to deallocation by the host

- constant and texture memory
  - the host can read and write
  - all threads (GPU) can read only

When this CUDA model is mapped to actual GPU hardware, one thread block is executed on one stream multiprocessor (see Figure 3) and does not migrate. Several blocks can reside concurrently on one multiprocessor.

A simple example of matrix addition is shown in code in Figure 11. The example contains ANSI C code for CPU and its equivalent for GPU using CUDA extension. The kernel function is defined with `__global__`, while all other input parameters are the same for general C function and CUDA kernel function.

Parallelization is done by unfolding the two `for` loops, where one loop indexes the rows (variable i) of the matrix and the second indexes the columns (variable j). In the kernel, variable i is calculated from the x direction (`blockIdx.x * blockDim.x + threadIDx.x`) while variable j is calculated from the y direction (`blockIdx.y * blockDim.y + threadIDx.y`). All the threads then read their two values from memory; add these two values and store the result back into memory. In the complete program, arrays a, b and c in the global GPU memory must be first allocated and data must be transferred from the host to GPU, then the kernel can be executed. The grid and block configuration is done before kernel execution in the main function.

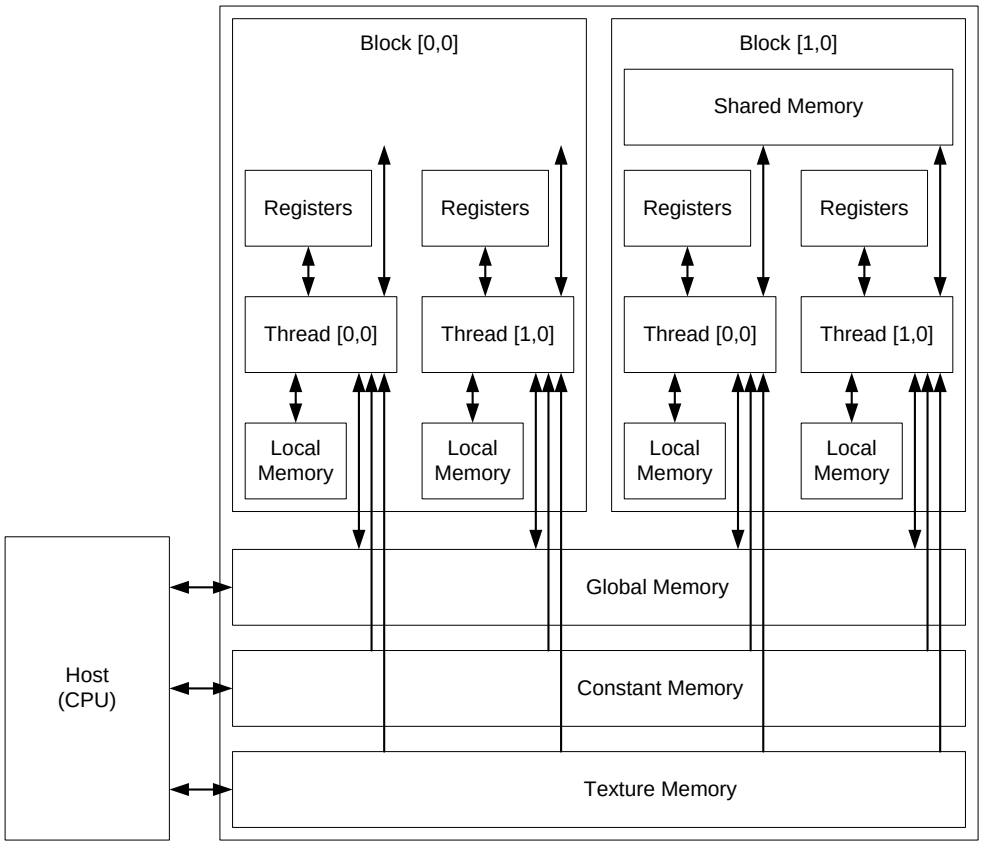

Figure 10: CUDA device memory space diagram

C program for CPU

```
void add_matrix
  ( float* a, float* b, float* c, int N ){
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
  }

int main() {
  add_matrix( a, b, c, N );

}
```

C program with CUDA extension for GPU

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
  c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);

}
```

Figure 11: Simple example of matrix addition for CPU and CUDA

5. Monitoring Pressure Vessels using Acoustic Emission

The acoustic emission (AE) testing method is a unique nondestructive testing (NDT) method where the material being inspected generates an acoustic signal that warns of impending failure [23, 24]. This method

relies on the detection of elastic stress waves produced by a sudden release of energy in the material [25]. The energy is released when a defect propagates, corrosion is formed, plastic zones are formed etc., as a result of the stress that is applied to the tested structure. The AE test method detects, locates, identifies, and displays flaw data for the stressed object the moment the flaw is created or extended. Real-time data processing is therefore essential.

Today's AE measurement systems are designed to record acoustic emission signal waveforms in addition to feature extraction used for real time evaluation. One of the areas where waveform based processing can significantly increase the precision of measurement results is AE events localization (for example, the localization method based on cross-correlation of AE signal envelopes). Since the oscillations of an AE signal are usually generated by AE sensors at their resonance frequency, the actual shape of the acoustic wave sensed by a sensor is the envelope of the oscillating signal. There are several known methods for calculation the envelope, e.g., by squaring the input signal and filtering it using a low-pass filter, or by using the Hilbert transform [26, 27].

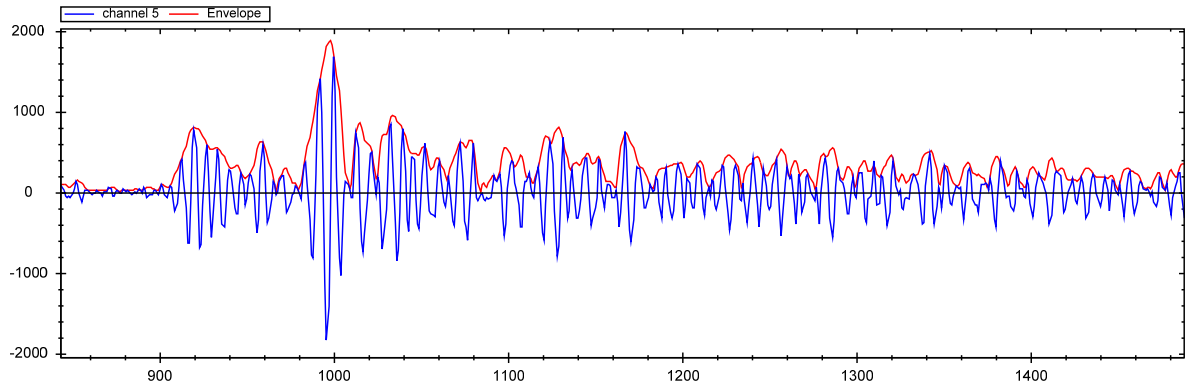

Figure 12: AE event waveform and its envelope calculated using the Hilbert transform

The Hilbert transform of the signal is calculated from the analytical signal, which can be found using the Parks-McClellan FIR filter [28] or the Fourier transform [29] and necessary operations in the frequency domain.

6. Accelerating of AE Signal Preprocessing Algorithms

This section describes GPU acceleration of following AE preprocessing algorithms: localization of AE events; envelope calculation; and filtering using a reduction filter. The position of AE events is significant information for similarity calculation (described in section 7). The signal envelope is used by the AE reduction filter.

6.1. Localization of AE Events

To calculate the position of an AE event, it is necessary to perform two operations: detect the time of the hit for the first three sensors, and calculate the hyperbolic triangulation to obtain the event location. Because the first operation compares all the samples in all channels with a threshold level, it is a suitable task for GPU. On the other hand, the calculation of hyperbolic triangulation processes only a few input parameters, and it is therefore calculated by CPU.

The start time of the event for the selected channel is detected as an index of the first sample higher than the threshold level. The kernel needs to search all channels and try to find the time of the hit in at least three of them before the hyperbolic triangulation in CPU is calculated.

To create an optimal kernel for the GPU architecture, it is necessary to use the principle of parallel reduction. Before parallel reduction is used, it is necessary to compare all the elements of the input array with the threshold level, which is either a constant (different for each channel) or a value calculated from the mean of the AE signal envelope in the current channel.

During the comparison part, the kernel copies the input data from global memory to shared memory. Each block copies the data into its local shared memory so that all threads in a block can access this part of the data. Next, the kernel compares the value of each element in the shared memory with the threshold level for a particular channel. If the threshold level is lower than value x_i of element i then the value of x_i is overwritten by index i (i is the index of the sample in a channel). Otherwise value x_i is set to the maximum value of the float type. When this process is completed, the shared memory contains the indexes of all samples higher than the threshold value.

The second part of the kernel uses a modification of parallel reduction to find the smallest index in the shared memory. The kernel must reduce 8×8192 samples to 8×1 sample. Since the CUDA model allows only 512 threads in a block which can be synchronized (there is no inter-block synchronization), and one kernel can only reduce 65535×512 elements to 65535×1, the reduction must be executed by two kernels.

The performance measurement shows that the first kernel needs 41.9μs to reduce 65535 elements to 512. The second kernel takes 7.5μs to reduce 512 elements to the final number of 8 elements.

6.2. Envelope Calculation Using the Hilbert Transform

The envelope of the acoustic emission signal represents the shape of the acoustic wave. The envelope is also used to improve the accuracy of AE events localization. This section describes the GPU acceleration of the envelope calculation using an analytic signal calculated by the Hilbert transform.

The analytic signal has one-sided frequency spectra and the negative frequencies are equal to 0. To approximate the analytic signal, the kernel needs to perform the following steps: calculate the FFT of the input sequence; replace the FFT coefficients that correspond to negative frequencies with zeros, and calculate the inverse FFT of the result. In addition, since the kernel works in the frequency domain, it

can easily perform low pass filtering by zeroing frequency coefficients higher than the cutoff frequency. The Hilbert transform can be approximated by the algorithm shown in Figure 13.

1: Calculate the forward FFT of the input sequence (vector x)
2: Create a vector h where elements $h(i)$ have the following values:
1 – for $i = 1, \frac{n}{2}+1$
2 – for $i = 2, 3, ..., \frac{n}{2}$
0 – for $i = \frac{n}{2}+2, ..., n$ where n is the length of the FFT transform
3: Calculate the element-wise product of vectors x and h
4: Optional - Perform low pass filtering – not part of the Hilbert transform algorithm
5: Calculate the inverse FFT of the sequence obtained in the step

Figure 13: Algorithm of Hilbert transform calculation using FFT

The CUFFT library [30], which is used to calculate forward and inverse FFT, is able to calculate multiple FFTs from one input array. In this case the FFT function is configured to perform 8 real to complex FFTs, where each FFT calculates the frequency domain representation of one channel (step 1 of the algorithm). The second, third and optionally fourth step of the algorithm are executed by the `HilbertInFFT` kernel. The last kernel of envelope calculation converts the complex output of inverse FFT to real by calculating the magnitude of the complex number to produce the envelope of the signal.

Hilbert transform	GPU	CPU total	Speedup [-]
Total time [μs]	384.1	12000	31.2

Table 3: The envelope calculation using Hilbert transform performance comparison

The CUFFT calculates 65535 long real to complex forward FFT using 4 kernels in 165.8 μs. The `HilbertInFFT` kernel processing time is 32 μs. The inverse FFT is again calculated by four kernels and takes 159.3 μs. The conversion from complex to real array requires 24.3 μs. The total execution time of all kernels is 384.1 μs. CPU needs 12 ms and is 31.2 times slower.

6.3. Acoustic Emission Reduction Filter

The acoustic emission reduction filter is used to remove events which clearly are not related to crack propagation. These events are caused, for example instance by small shifts of the monitored structure and

therefore are irrelevant for future detection of destructive AE sources. Their main properties are: the event duration is long (one event may be recorded as multiple events), their maximum amplitude is very high (very often higher than the range of the AD converter), and the rise time is also very long.

The AE decimation filter processes the envelope of the AE event. In the first step it calculates the mean value of the envelope and multiplies it by a constant C (set by the operator) to get the threshold level t of the filter. In the second step, the filter decimates the envelope of the signal by factor f. This means that every f samples of the envelope are substituted by their mean value. In the last step, the algorithm counts how many samples of the decimated envelope are higher than t. If there are no samples higher than the threshold the event is not valid in a channel. If at least one sample is higher than the threshold t, the event is valid. The AE event is valid if at least three channels pass the filtration test.

AE decimation filter	GPU[μs]	CPU[μs]	Speedup[-]
Signal Decimation kernel	41.7	-	-
Signal Decimation Filter kernel	9.7	-	-
Total time	51.4	165	3.2

Table 4: Comparison of the AE decimation filter performance for CPU and GPU

The GPU implementation of the decimation filter contains two kernels. The first kernel (Signal Decimation) decimates the input sequence by a factor of f (using the mean value of the decimated samples), and the second kernel (Signal Decimation Filter) calculates the mean value of the decimated signal in each channel, compares the decimated signal with the filtration threshold f, and counts how many samples of the decimated signal are higher than f.

The execution is 41.9 μs per event for the first kernel and 9.7 μs for the second kernel. When compared with the execution time on CPU, which is 166 μs, GPU is 3.2 times faster than CPU.

7. Accelerating AE waveform similarity calculation

Acoustic emission filtration methods use the similarity of AE event waveforms to locate the AE sources related to the destructive processes in the material. A significant problem that arises during long term monitoring is the large amount of data (tens of GBytes) recorded by the AE measurement system.

The similarity calculation process is displayed in Figure 14. The input is a set of all AE events recorded during the monitoring session.

In the first step, all events that are not part of any cluster (all AE events which have no or only a small number of neighbors) are removed from the set. In the second step, the set of all remaining AE events is divided into clusters on the basis of its position using agglomerative hierarchical clustering and a distance matrix. Removing unclustered events and dividing the events into location clusters significantly reduces the

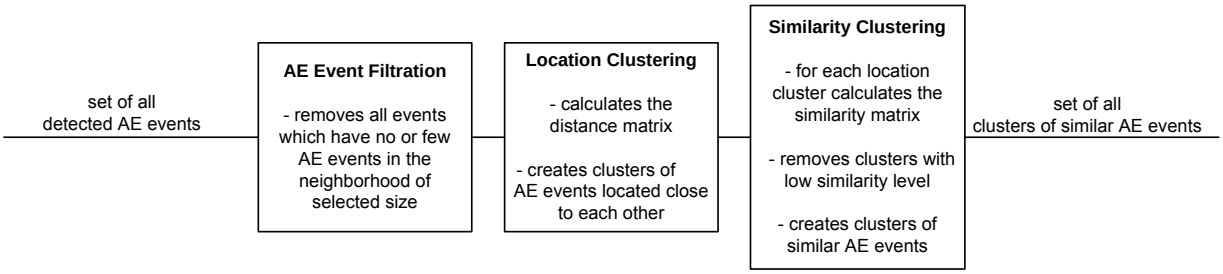

Figure 14: Main blocks of the similarity calculation algorithm accelerated by GPU

processing time, because the similarity is calculated (third step) only among the AE events from the same location cluster.

The output of the algorithm is a set of clusters with similar AE events only. All the clusters which do not contain any similar events are removed, as well as all dissimilar AE events, from clusters which contain some similar events.

7.1. Filtering Unclustered AE Events

The first step in similarity calculation is to filter AE events which are not located in an area with a high concentration of AE events. By filtering out the unclustered events the computation time necessary for generating the distance matrix is reduced.

The algorithm calculates how many events in the neighborhood of the selected event are of a predefined size. This way the algorithm processes every event in the set. The input values for this algorithm are: the positions of all AE events (x and y coordinates); neighborhood size; and the minimum number of neighbors. If an event does not have enough neighbors it is removed from further processing.

The parallelization strategy is that each thread calculates the number of neighbors of one event. When the kernel starts, the thread with `thread.ID` equal to m loads the position of AE events e_m into the register. Then the thread loads the position of one event after another into the loop (let this AE event be called $e_{m|n}$, where n is index of the event compared in the loop), and calculates the Euclidian distance between events e_m and $e_{m|n}$. If the distance between these two events is smaller than the predefined value, the number of neighbors of event e_m is increased by one. The output of the kernel is an array where the n-th element contains the number of neighbors of the n-th event.

	GPU Execution time [s]	CPU execution time [s]	Speedup
Filtration of unclustered events - input set size 8500 AE events	0.18	12	66

Table 5: Performance comparison between CPU and GPU for filtration of unclustered AE events

```
extern "C" __global__ void CountNeighbors(float* xPos, float* yPos, int* neighCountArray,
float neighborhoodSize)
{
        unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int tid = threadIdx.x;

        float dx, dy, dx2, dy2;

        if (i < count) {
                dx = xPos[i];
                dy = yPos[i];
                for (int x = 0; x < count; x++) {
                        if (x == 0) neighCountArray[i] = 0;

                        dx2 = (dx - xPos[x]) * (dx - xPos[x]);
                        dy2 = (dy - yPos[x]) * (dy - yPos[x]);

                        if (sqrt(dx2 + dy2) < neighborhoodSize) neighCountArray[i]++;
                }
        }
}
```

Figure 15: Kernel which calculates how many neighbors each AE event has

When the algorithm is executed on CPU the processing time is 12 s if 8500 AE events are processed. The GPU execution time for the same number of AE events is 0.18 s, which means that the speed up is 66 when data is processed using GPU.

7.2. Calculating the Distance Matrix

The distance matrix is calculated for all events which are located in areas with a high concentration of AE events and therefore passed the filtration from the previous section. The distance matrix contains the Euclidian distances between all these events. The reason why distance matrix is not calculated as a partial output of the kernel from the previous section is the size of the output array. If the output array of the filtration kernel is N elements long, where N is the total number of AE events, the distance matrix will have N^2 elements. For example, if the filtration reduces the number of processed events from 8500 to 5100 then the size of the distance matrix is reduced from $72.25 \cdot 10^6$ to $26.01 \cdot 10^6$ elements, which is a significant reduction of GPU memory usage. Since GPU works with a float type which is 4 bytes long, the array size necessary to store the distance matrix is reduced from 289MB to 104MB.

The performance of the distance matrix kernel is shown in Table 6. If 5100 AE events are processed, CPU needs 8.1 s to calculate the distance matrix, while GPU is 36 times faster and needs only 0.22 s.

	GPU Execution time [s]	CPU execution time [s]	Speedup
Calculation of distance matrix, input set size 5100 AE events	0.22	8.1	36

Table 6: Performance comparison between CPU and GPU for distance matrix calculation

7.3. Agglomerative Hierarchical Clustering

Agglomerative hierarchical clustering detects clusters on the basis of data from the distance or similarity matrix. This method was chosen because the clustering process can be stopped if the distance between two cluster centroids is higher than a defined value, which is a practical condition for clustering of AE events since the operator can define the maximum diameter of the cluster.

Two methods are suitable for AE clustering: the Unweighted Pair Group Method (UPGM), which minimizes the amount of distortion between the dendrogram and the similarity or proximity matrix (used for location clustering); and the Weighted Pair Group Method (WPGM), which gives equal weights when computing fusion similarities to the two branches of the dendogram that are about to fuse (used for similarity clustering) [31].

1: Compute the proximity or similarity matrix
2: **repeat**
3: Merge the closest two clusters
4: Update the proximity matrix to reflect the proximity between the new cluster and the original clusters
5: **until** Only one cluster remains

Figure 16: Basic agglomerative hierarchical clustering algorithm

7.4. Calculating the Power Spectrum Matrix

The similarity of AE events is calculated in the frequency domain. Since the FFT of the AE event waveforms was calculated as part of signal envelope estimation it is not necessary to calculate it again. Instead, the kernel which calculates the signal envelope is modified to calculate the magnitude of the frequency spectra together with the signal envelope. The modified kernel (shown in Figure 17) has an additional output array called `powerSpectra`, where the magnitudes of the frequency spectra calculated by `ComplexToRealMagnitude` function are stored.

Before the similarity is calculated, the waveforms of all AE events must be scaled so that the maximum amplitude is equal to 1. The reason for scaling is the different amount of energy released by the AE source and converted to an acoustic wave, which affects the maximum amplitude of the AE signal.

This operation requires knowledge of the maximum amplitude of the signal, which is found by two kernels (`FindAbsMax1` and `FindAbsMax2`). These two kernels are modifications of parallel reduction designed to find the maximum value of the vector.

```
extern "C" __global__ void HilbertInFFT(Complex* a, Complex* out,float* powerSpectra,
unsigned int size, float scale, unsigned int LPFilter, unsigned int offset)
{
	__shared__ Complex shared[512];
	__shared__ float   sharedSpectra[512];

	const unsigned int numThreads = blockDim.x * gridDim.x;
	const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
	const unsigned int tid = threadIdx.x;

	unsigned int iO = i % (offset / 2);

	Complex c0; c0.x = 0; c0.y = 0;
	Complex c1; c1.x = 1; c1.y = 0;
	Complex c2; c2.x = 2; c2.y = 0;

	if (i < (size/2)) shared[tid] = a[i];
	__syncthreads();

	if (i < (size/2)) {
		Complex tempInp = shared[tid];
		float absSpect = ComlexToRealMag(tempInp);
		sharedSpectra[tid] = absSpect;

		if(iO > 0 && iO < (offset/2))
			shared[tid] = ComplexScale(ComplexMul(tempInp, c2), scale);

		if(iO == 0 || iO == (offset/2))
			shared[tid] = ComplexScale(ComplexMul(tempInp, c1), scale);

		if(iO > (offset / 2) || iO > LPFilter)
			shared[tid] = c0;

	}
	__syncthreads();

	if (i < (size/2)) {
		Complex c; c.x = i; c.y = 0;
		out[((i / (offset / 2)) * offset) + 0 + (i % (offset / 2))] = shared[tid];
	}
	__syncthreads();

	if (i < (size/2))
		out[((i / (offset / 2)) * offset) + (offset / 2) + (i % (offset / 2))] = c0;
	__syncthreads();

	if (i < (size/2)) powerSpectra[i] = sharedSpectra[tid];
}
```

Figure 17: Kernel which calculates the signal envelope and the magnitude of the frequency spectra

All processing described up to this point was done during the data preprocessing part, and therefore the power spectra were transferred to the system memory to free the GPU memory for further processing. The data transfer between GPU and CPU is the performance bottleneck. Scaling and the last part of the power spectrum calculation is therefore done by CPU.

The power spectrum is calculated according to the following formula:

$$S_{xx}(k) = \frac{|X(k)|^2}{N^2}, \tag{1}$$

where $|X(k)|$ is the magnitude of the frequency spectra, which is provided by kernel `HilberInFFT`, and N is the length of FFT.

Since scaling of the vector also needs to be done, formula (1) is modified to:

$$S_{xx}(k) = \frac{\left(\frac{1}{A_{\max}} |X(k)|\right)^2}{N^2}, \tag{2}$$

where $A_{\max}$ is the maximum amplitude of the waveform used for scaling (provided by `FindAbsMax` kernels).

The last step is to create the power spectra matrix which contains the power spectra of all AE events in one location cluster:

$$M_{Sxx} = \begin{pmatrix} S_{xx}(k)_1 \\ S_{xx}(k)_2 \\ \\ S_{xx}(k)_L \end{pmatrix}, k = 0, 1, ... \frac{N}{2} + 1 \tag{3}$$

where $M_{Sxx}$is power spectra matrix, L is number of AE events in the cluster, and N is length of FFT.

The power spectrum matrix is used to calculate the similarity matrix. This step is described in the next section.

7.5. Calculating the Similarity Matrix

The similarity matrix is a matrix of scores which express the similarity between two AE events. The element $d_{a|b}$ of the similarity matrix describes the similarity between event e_a and event e_b.The similarity matrix is calculated from the power spectrum matrix, where row number b contains the power spectrum of AE event number b in a cluster.

Calculation of the similarity is computationally expensive, because to get the similarity of events $e_a$and e_b GPU needs to sum all differences between two rows as shown in formula (4). If the cluster contains L events, GPU must execute equation (4) L^2 times.

$$d_{a|b} = 2 \cdot \sum_{k=0}^{\frac{N}{2}+1} \left| S_{xx}(k)_a - S_{xx}(k)_b \right|, \tag{4}$$

where $d_{a|b}$ is difference between events $e_a$and e_b, and $S_{xx}(k)_a, S_{xx}(k)_b$ are power spectra of events e_a and e_b.

The parallelization strategy used for computing the similarity matrix is: calculate the similarity of one selected event with all the rest of the events at a time. This strategy is executed by two kernels (`GetSimVector` and `ReduceSimVector`) which are executed L times to calculate all rows of the similarity matrix.

Calculation of Similarity Matrix number of events to process[-]	GPU Execution time [s]	CPU execution time [s]	Speedup
100 events	0.05	4.1	82
210 events	0.19	11.2	58.9
410 events	0.74	36.5	49.3
650 events	1.78	88.6	49.7
1200 events	6.27	288.4	45.9

Table 7: Performance comparison between CPU and GPU for similarity matrix calculation

The performance measurements show that the speed up is from 46 to 82, depending on how many AE events are compared. The similarity matrix calculation time is up to 6.3 s when the similarity of 1200 AE events is calculated. In most cases, the number of AE events in the cluster is from 100 to 400, which means that the calculation time is no more than 0.74 seconds.

8. Results

The algorithms presented here were accelerated by Nvidia Quadro FX3700 with 112 CUDA processors (theoretical performance 430 GFLOPS in single precision), 512 MB of RAM (51.2 GB/s memory bandwidth). The CPU of the workstation is Intel Core2Quad Q6600@2.4GHz with 9.6 GFLOPS a core (total 38.4 GFLOPS in double precision).

8.1. Signal Preprocessing Results

In this section, we present the performance measurement of the preprocessing part (event localization, envelope calculation of the AE signal and AE reduction filter). This whole preprocessing block consists of 15 kernels, out of which 8 kernels are part of the CUFFT library and were executed during forward and inverse FFT.

The whole preprocessing block is much faster when GPU acceleration is used. Table 8 shows that GPU is from 14 to 47.9 times faster than CPU (depending on how many channels are processed).

Another important value is the execution time of an event. GPU works all the time with a fixed length of 8192 samples a channel, which is equal to a 5.2 ms long AE event. Table 8 shows that GPU is able to process 32 channels in 4.3 ms (which is still faster than is required for real time processing), and 64 channels in 6.9 seconds, which is still fast enough for a real-time AE system.

Event executiontime	GPU[ms]	CPU[ms]	Speedup GPU over CPU [-]
8 channels	2.0	28-42	14-21
16 channels	2.9	54-83	18.6-28.9
32 channels	4.3	110-163	25-37.9
64 channels	6.9	219-331	31.7-47.9

Table 8: Total execution time necessary to process one AE event by GPU and GPU

8.2. Similarity Calculation Results

The similarity calculation consists of three main tasks: filtering unclustered events, location clustering and similarity clustering. The clustering itself is executed by CPU, while both matrices are calculated by GPU.

	Number of AE events processed [-]	GPU execution time [s]	CPU execution time [s]	Speedup
Filtration of unclustered events	8500	0.18	12	66
Calculation of distance matrix	5100	0.22	8.1	36
Calculation of Similarity Matrix	100	0.05	4.1	82
	210	0.19	11.2	58.9
	650	1.78	88.6	49.7
	1200	6.27	288.6	45.9

Table 9: Processing times of similarity calculation algorithms

The filtration of unclustered events is 66 times faster if executed by GPU when 8500 AE events are processed. CPU needs 12 s, while GPU processes the same amount of data in 0.18 s. The calculation time of distance matrix is reduced 36 times when accelerated by GPU. The distance matrix size is 5100×5100, and GPU needs 0.22 s while the CPU processing time is 8.1 s. The last and most computationally expensive task is calculating the similarity matrix. Here, GPU is from 45.9 to 82 times faster than CPU, depending on the size of the similarity matrix. Calculating the similarity matrix for a cluster containing 1200 AE events is reduced from 288.6 seconds to 6.3 seconds by GPU acceleration.

9. Conclusion

This paper proposes acceleration of signal processing methods for acoustic emission and acceleration of similarity calculation using GPU. The advantages of GPU-based accelerators are: low cost; high performance; and the fact that they can be used in any industrial workstation or laptop and are therefore suitable for onsite AE monitoring.

The algorithms described in this paper have been developed for offline processing of AE signal waveforms recorded during long term AE monitoring sessions.

The signal preprocessing methods process each recorded AE event only once, and are therefore more dependent on the transfer rate between system memory and GPU memory. This factor slows down overall GPU performance, but the results still prove that GPU is on average 32 times faster than CPU when performing the following operations: AE event localization, envelope calculation and filtration with an AE reduction filter. These three algorithms form the basic GPU framework for AE signal processing.

Filtration techniques based on similarity of AE events are less dependent on the transfer rate between system and GPU memory, because each AE signal waveform transferred to GPU memory is processed N times, where N is the number of events in the cluster among which the similarity is calculated. This model is even more suitable for GPU architecture, and the processing speed of similarity matrix calculations is therefore up to 82 times faster than CPU. For the same reason, the communication overhead also affects algorithms less for filtration of unclustered events and calculation of distance matrix. Their respective GPU processing times are 66 and 36 times faster.

The performance of our CUDA implementation was tested on Nvidia Quadro FX3700 with 112 CUDA processors, which has a theoretical performance of 430 GFLOPS in single precision and has 512 MB of RAM. The processing speed can be significantly improved using Nvidia Tesla C1060 HPC accelerator with 4GB of RAM and 240 CUDA cores (933 GFLOPS in single precision and 78 GFLOPS in double precision) or using the latest generation Nvidia Fermi Tesla C2050 cards with 448 CUDA cores (1030 GFLOPS in single precision and 515 GFLOPS in double precision) [32, 33].

References

[1] International Technology Roadmap for Semiconductors, International technology roadmap for semiconductors - system drivers, [Online] (2007).
URL http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_ System-Drivers.pdf

[2] G. Blake, R. Dreslinski, T. Mudge, A survey of multicore processors, Signal Processing Magazine, IEEE 26 (6) (2009) 26–37. doi:10.1109/MSP.2009.934110.

[3] Intel, Intel 64 and IA-32 Architecture Software Developer's Manual, Vol. 3A, 2008.

[4] NVIDIA Corp., NVIDIA CUDA: Compute unified device architecture, 2008.

[5] Advanced Micro Devices Inc., ATI Radeon HD4870 & ATI Radeon HD4850 - GPU Specification, 2008.

[6] IBM, Cell broadband engine architecture, [Online] (2007).
URL https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs /1AEEE1270EA2776387257060006E61BA/$file /CBEA_v1.02_11Oct2007_pub.pdf

[7] H. P. Hofstee, Power efficient processor architecture and the cell, in: Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11), 2005.

[8] F. Franchetti, M. Puschel, Y. Voronenko, S. Chellappa, J. Moura, Discrete Fourier transform on multicore, Signal Processing Magazine, IEEE 26 (6) (2009) 90 –102. doi:10.1109/MSP.2009.934155.

[9] NVIDIA Corp., Nvidia's next generation cuda compute architecture fermi, [Online] (2009).
URL http://www.nvidia.com/object/IO_86776.html

[10] NVIDIA Corp., CUDA Reference Manual, Santa Clara, CA, 2010.

[11] Advanced Micro Devices Inc., Ati stream computing - technical overview, [Online] (2009).
URL http://developer.amd.com/gpu_assets/ Stream_Computing_Overview.pdf

[12] A. Munshi, The OpenCL specification version 1.0, Khronos Group, Beaverton, OR, 2009.

[13] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for gpus: stream computing on graphics hardware, in: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, ACM Press, New York, NY, USA, 2004, pp. 777–786.

[14] The Mathworks, Parallel computing toolbox - documentation, [Online] (2009).
URL http://www.mathworks.com/access/helpdesk/help/toolbox/ distcomp/

[15] Interactive Supercomputing Inc., Starp for matlab users, [Online] (2009).
URL http://www.interactivesupercomputing.com/ products/starpandmatlab.php

[16] N.T. Bliss, J. Kepner, pmatlab: Parallel matlab toolbox, [Online] (2009).
URL http://www.ll.mit.edu/mission/isr/pmatlab/pmatlab.html

[17] Blue Collar Computing Software team., bcmpi, [Online] (2009).
URL http://www.osc.edu/bluecollarcomputing/applications/ bcMPI/index.shtml

[18] NVIDIA Corp., Matlab plug-in for cuda, [Online] (2009).
URL http://www.nvidia.com/object/matlab
cuda.html

[19] The GP-you Group, Gpumat: Gpu toolbox for matlab, [Online] (2009).
URL http://gp-you.org/

[20] T.-X. Corp, Gpu-accelerated computing for very high-level languages, [Online] (2009).
URL http://www.txcorp.com/products/GPULib/

[21] AccelerEyes, Jacket v 1.3, [Online] (2010).
URL http://www.accelereyes.com/

[22] S. Samsi, V. Gadepally, A. Krishnamurthy, Matlab for signal processing on multiprocessors and multicores, IEEE Signal Processing Magazine 27 (2) (2010) 40–49. doi:10.1109/MSP.2009.935421.

[23] P. E. Mix, Introduction to Nondestructive Testing, 2nd Edition, John Willey & Sons, Hoboken, NJ, 2005.

[24] R.K. Miller, E. v. K. Hill, Acoustic Emission Testing, 3rd Edition, Vol. 6, American Society for Nondestructive Testing, Columbus, OH, 2005.

[25] A. A. Anastasopoulos, D. A. Kourousis, P.T. Cole, Acoustic emission inspection of spherical metallic pressure vessels, in: 2nd International Conference on Technical Inspection and NDT, Tehran, Iran, 2008.

[26] T. D. Rosing et al, Springer Handbook of Acoustics, 3rd Edition, Springer Science, New York, NY, 2007.

[27] C. U. Groose, M. Ohtsu, Acoustic Emission Testing, Springer-Verlag, Ed. Berlin, Germany, 2008.

[28] J. G. Proakis, D. G. Manolakis, Digital Signal Processing : Principles, Algorithms, and Applications, 3rd Edition, Prentice Hall, Upper Saddle River, NJ, 1996.

[29] R. Bracewell, The Fourier Transform and Its Applications, 2nd Edition, McGraw-Hill, 1986.

[30] NVIDIA Corp., CUFFT Library, Santa Clara, CA, 2007.

[31] M. Steinbach, V. Kumar, P.-N. Tan, Introduction to Data Mining, Addison-Wesley, 2006.

[32] NVIDIA Corp., Nvidia tesla c2050/c2070 gpu computing solution for workstations, [Online] (2009).
URL http://www.nvidia.com/docs/IO/43395/ NV_DS_Tesla_C2050_C2070_Final_lowres.pdf

[33] NVIDIA Corp., Tesla c1060 computing processor, [Online] (2010).

URL http://www.nvidia.com/object/product_tesla_c1060_us.html